Sveučilište u Zagrebu
Fakultet elektrotehnike i računarstva
Zavod za elektroniku, mikroelektroniku, računalne i inteligentne sustave

Web Architecture, Protocols, and Services

Arhitektura, protokoli i usluge weba

UNIZG-FER 222464

Remote Procedure Calls - RPC

Klemo Vladimir, 2023.

IPC – Interprocess communication

- files on disk and memory-mapped files
- shared memory
- signals
- sockets
- pipes
- messages
  - RPC, RMI, etc.

Web service

- The W3C defines a *Web service* generally as:
  - a software system
  - designed to support interoperable machine-to-machine interaction
  - over a network
- In a 2004 document, the W3C extended the definition – 2 basic types of web services:
  - (1) **REST-compliant Web services**, in which the primary purpose of the service is to manipulate representations of Web resources using a uniform set of stateless operations.
  - (2) **Arbitrary Web services**, in which the service may expose an arbitrary set of operations.
    - RPC

RPC - Introduction

- Late 60s, early 70s.
  - *J. E. White, A High-Level Framework for Network-Based Resource Sharing. RFC 707 (1975)*
  - *P. B. Hansen, Distributed processes (1978)*
    - *“I now understand that it was really a small operating system, I had programmed. However, in the mid 1960s, the dividing line between language implementation and operating systems was still not clearly understood.”*
- Bruce Jay Nelson (Xerox) is generally credited with coining the term "Remote Procedure Call" (1981)
- First implementation Xerox 1981.
  - Lupine/Courier system
- SUN RPC 1984
  - Network File System
- Main functionality
  - ***normalizes the method-call semantics between systems residing either in the same address-space or in remote address-spaces***

RPC – Introduction (2)

- Request/response message passing protocol
  - allows implementation of client/server systems
  - **synchronous** (blocking call) and **asynchronous** (non-blocking call)
- The invocation of the remote service
  - appears as a normal procedure call
- OOP
  - RMI – Remote Method Invocation
- Separates interface and implementation
  - abstract interface declaration → portability
- Disadvantage
  - Less reliable
  - Slower (1-2 orders of magnitude) than local call
- RPC can be built into language/platform
  - Erlang

RPC – Basic workflow

- **1.** Service is described using some form of **Interface Definition Language (IDL)**
  - Sun RPC: RPC language
  - gRPC: Protobufs
- **2.** Special program takes IDL on input and produces client/server stubs (or proxies)
  - Sun RPC: rpcgen protocol compiler
  - gRPC: grpc.tools.protoc
- **3.** Client program uses a local procedure call into the client stub
  - provides the same signature as the service itself
- **4.** Client stub transparently communicates the service's parameters to the server program by sending an RPC request

RPC – Basic workflow (2)

- **5.** Data is encoded using some marshalling/ serialization format
  - Over some kind of transport: TCP/UDP/HTTP/...
  - Sun RPC: External Data Representation, XDR
  - gRPC: protobuf format
- **6.** On the server side, this request is extracted by the server stub
  - again, performs a local procedure call into the user-provided service implementation
- **7.** Service's result is then returned the same way

RPC – Historical overview

- 1980s
  - C/C++ Unix RPC, EDI using ASN.1, ...
- 1990s
  - DCOM, CORBA, JavaRMI, ...
- 2000s
  - Web, HTTP, REST, …
- 2010s
  - “modern” RPC

1. generation RPC

- **Sun/ONC RPC**
  - 1984.
  - Serialization
    - External Data Representation (XDR)
      - IETF standard 1995.
      - Base unit of 4 bytes
        - boolean, int, float, double, structure, enum, string, union, ...
    - TCP/UDP
  - IDL for interface definition
  - Does not support OO features like polymorphism, exceptions, etc.

1. generation RPC

- **Sun RPC**
  - `rpcgen -a -C add.x`
  - Creates:
    - client (add_client.c)
    - server (add_server.c)
    - Makefile

```
struct intpair {
        int a;
        int b;
};

program ADD_PROG {
        version ADD_VERS {
                int ADD(intpair) = 1;
        } = 1;
} = 0x23451111;
```

  - Full example
    - https://www.cs.rutgers.edu/~pxk/417/notes/rpc/index.html

2. generation RPC (object-oriented)

- **CORBA**
  - 1991.
  - Common Object Request Architecture
    - 1991. (C), 1997. (C++), 1998. (Java)
  - OMG (Object Management Group) Consortium
  - OS/language/network independent
  - paradigm: request services of a **distributed object** (RMI)
  - client does not have to be object-oriented
  - IDL for interfaces
  - CDR (Common Data Representation) as serialization format (binary)

2. generation RPC (object-oriented)

- **CORBA**
  - objects are identified by references
  - ORB - Object Request Broker for RPC
    - delivers requests to the object and returns results to the client
  - specification addresses data typing, exceptions, network protocols, communication timeouts, transactions, etc.
  - **Standardized, open, platform independent**
  - **Complex with many implementation problems and bad governance**

2. generation RPC (object-oriented)

- **CORBA**
  - omniORBpy

```
// echo_example.idl
module Example {
    interface Echo {
        string echoString(in string mesg);
    };
};


$ omniidl -bpython example_echo.idl


http://omniorb.sourceforge.net/omnipy3/omniORBpy/omniORBpy002.html
```

2. generation RPC (object-oriented)

- **MS DCOM**
  - Microsoft response to CORBA
  - 1995.
  - Extends
    - OLE (Object Linking and Embedding)/COM
    - DCE RPC to allow objects to communicate between machines
  - C++ implementation generates client proxy and server stub from the IDL
  - Language neutral, object-oriented
  - MS proprietary
    - (more open) CORBA as major competitor

2. generation RPC (object-oriented)

- **Java RMI**
  - 1995.
  - extension for Java called Remote Method Invocation
  - Architecture
    - Client
    - Server
    - Object Registry
  - messages are serialized Java classes
  - rmic, compiler for RMI stubs
  - rmiregistry
  - Java only (unlike CORBA), no IDL
  - Tutorial: https://docs.oracle.com/javase/tutorial/rmi/index.html

3. generation RPC (XML-based)

- Motivation
  - DCOM and CORBA use binary format
  - **Firewall issues**
  - Reuse **XML** and **HTTP** expertise and tools
    - XML ~1996.
    - Standards-based, platform-independant
    - Immune to firewall (text, HTTP port 80)

3. generation RPC (XML-based)

- **XML RPC**
  - 1998.
  - messages are "human-readable" XML
  - uses HTTP for transport
  - no official IDL compiler
  - simple specification
  - without much support from the industry
  - Spec: http://xmlrpc.scripting.com/spec.html

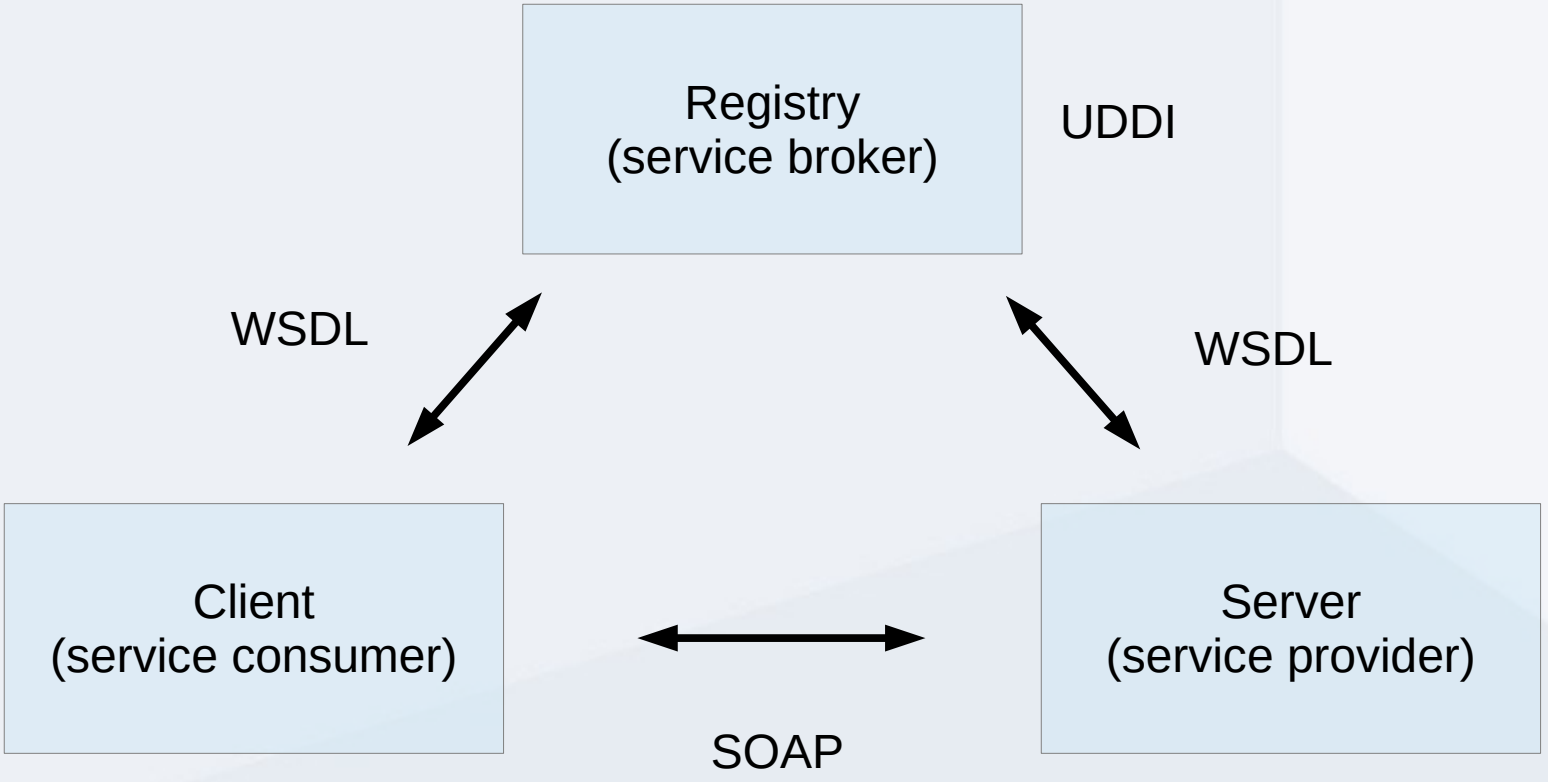
  - *See also: JSON RPC (2005.)*
    - *https://www.jsonrpc.org/specification*

3. generation RPC (XML-based)

- **SOAP**
  - 1999.
  - Simple Object Access Protocol
  - Evolved from XML RPC
  - Platform independant
    - XML, HTTP
  - Part of SOA stack (Service-oriented Architecture)
    - Clients, services, service registry
      - **SOAP** for communication/serialization
      - **WSDL** as interface definition language (service contract)
      - **UDDI** for registry/discovery
  - CORBA-like level of complexity

3. generation RPC (XML-based)

- **SOA**
  - Basic architecture



3. generation RPC (XML-based)

- **SOA**
  - **UDDI**
    - Universal Description, Discovery and Integration
    - XML-based registry of services
    - Not widely adopted
  - **WSDL**
    - Web Service Definition Language
    - XML-based interface description language
    - Basic elements
      - operation, message, types
      - interface, binding, endpoint, service
      - Example: https://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/#basic-example

3. generation RPC (XML-based)

- **SOAP**

```
<s:Envelope xmlns:s="...">
  <s:Body>
    <procedureName s:encodingStyle=“.../soap-encoding">
       <arg1 xsi:type=“TypeA">abc</arg1>
       <argN xsi:type=“TypeB">123</arg2>
    </procedureName>
  </s:Body>
</s:Envelope>
```

```
<s:Envelope xmlns:s="...">
  <s:Body>
    <procedureNameResponse s:encodingStyle=“.../soap-encoding">
       <return xsi:type="xsd:TypeA">...</return>
    </procedureNameResponse>
  </SOAP:Body>
</SOAP:Envelope>
```

3. generation RPC (XML-based)

- **MS .NET Remoting**
  - OLE → COM → DCOM → *.NET Remoting* → WCF
  - COM/DCOM was too low-level with explicit reference counting
  - supports SOAP for interoperability
  - also, binary support for performance
  - has proxy objects that act as representative of the remote objects and
    - channels for transporing messages to and from remote objects
  - MS response to Java RMI
  - Mostly .NET clients
    - like Java RMI had mostly Java clients

3. generation RPC (XML-based)

- **AJAX**
  - term from 2005.
    - **A**synchronous **J**avaScript **A**nd **X**ML
  - technology since 1999.
  - Usually called from browser’s JS engine
    - using XMLHTTPRequest
  - Main use case
    - dynamic web pages
  - XML is often replaced with **JSON**
    - http://www.json.org/xml.html

3. generation RPC (XML/JSON-based)

- **REST**
  - 2000.
  - web-resources oriented services
    - Based on Web/HTTP
    - No method call abstractions in style of RPC
      - Message is not method call but resource representation
    - All comunication must be stateless and cacheable
    - GET /users
    - GET /users/<user_id>
    - GET /users/<user_id>/photos
    - [RPC – get_users(), get_user(user_id), …]
  - RESTafarians
    - Anti-SOAP campaign led by Roy Fielding
      - Movement possibly related to anti-Microsoft which supported SOAP
    - Google 2006. dropped support for SOAP

4. generation RPC

- What is the problem with XML?
  - For example, *Protocol buffers* serialization format has many advantages over XML for serializing structured data. They:
    - are simpler
    - are 3 to 10 times smaller
    - are 20 to 100 times faster
    - are less ambiguous
    - generate data access classes that are easier to use programmatically

4. generation RPC

- **GRPC**
  - https://grpc.io/docs/what-is-grpc/introduction/ !!!
  - https://grpc.io/docs/tutorials/basic/python.html !!!
  - Open source RPC by Google, 2015.
    - HTTP/2, Authentication, Streaming, cross-platform bindings
  - IDL based on **Protocol Buffers**
    - Google, 2001., public 2008.
    - Protocol buffer data is structured as messages
      - series of name-value pairs called fields
      - **profo** file message example:

```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_ponycopter = 3;
}
```

4. generation RPC

- **gRPC**
  - Services are also defined in the proto files

```
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}



# client/server code for python is generated with:
$ python -m grpc_tools.protoc service.proto
```

4. generation RPC

- **gRPC**
  - Efficient IPC
    - Binary protocol on top of HTTP/2
    - Client and server-side streaming
    - Integrated with cloud-native systems
  - Simple service interface
    - Well-defined schema for contract-first development of services
    - Strongly-typed
    - Polyglot approach for multiple programming languages
  - Built-in features
    - Encryption, authentication, resiliency, service discovery, load balancing etc.

4. generation RPC

- **gRPC**
  - Scalar types
    - double, float
    - int32, int64, uint32, uint64, sint32, sint64
    - fixed32, fixed64
    - bool, string, bytes
  - Structured types as Messages
    - Each field has a unique number
      - used for binary transmission and they should not be changed
      - Numbers 1-15 use 1 byte
    - Field rules
      - singular, optional, repeated, map, oneof, reserved

4. generation RPC

```
/* from https://developers.google.com/protocol-buffers/docs/pythontutorial */

syntax = "proto2";
package tutorial;


message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;


  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }


  message PhoneNumber {
    optional string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }


  repeated PhoneNumber phones = 4;
}


message AddressBook {
  repeated Person people = 1;
}
```

4. generation RPC

- **Communication patterns**
  - **Unary RPC**

```
service OrderManagement {
    rpc getOrder(google.protobuf.StringValue) returns (Order);
}

message Order {
    string id = 1;
    repeated string items = 2;
    string description = 3;
    float price = 4;
    string destination = 5;
}
```

4. generation RPC

- **Communication patterns**
  - **Server-Streaming RPC**

```
service OrderManagement {
    rpc searchOrders(google.protobuf.StringValue) returns (stream Order);
}

message Order {
    string id = 1;
    repeated string items = 2;
    string description = 3;
    float price = 4;
    string destination = 5;
}
```

Also: Client-streaming and Bidirectional streaming supported

4. generation RPC

- **Apache Thrift**
  - 2007.
  - cross-language services development
  - different protocols (binary and textual)
  - different transport mediums (files, memory, sockets, …)

  - https://thrift.apache.org/static/files/thrift-20070401.pdf **!!!**